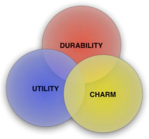


Class 12 SSW 565

Gregg Vesonder
Stevens Institute of Technology

Road Map

- Log
- Refactoring
- Reading: <u>Refactoring</u>, Fowler
- Reading next week, REVIEW!

Log

- “Mirror, mirror on the wall, who’s the fairest of them all?”

http://www.imdb.com/title/tt0029583/

Key Dates

- July 20th – take home final due July 23rd

Refactoring

- Refactoring is changing software system without altering external behavior, yet improves its internal structure
  - Risky (Brooks up to .5 probability of introducing additional errors)
  - Disciplined - dangerous if not
  - Design focused, more improving design than code tweaking
  - Reversing entropy – anti-regressive maintenance
  - Extreme programming - continuous design
  - Design patterns as targets for refactoring

Chptr 1 Video Store example

- Customer class statement routine (and different) is too long and contains all sorts of functionality
- For a Quick & Dirty program it may be acceptable because it does work.
- For added functionality to cut and paste or redesign? - issue of consistent fixes in all copies
- Versus "if it ain't broke don't fix it" but if it makes life difficult …
- So customer wants, HTML version, may change classification of videos, …

Begin Refactoring

- Need solid suite of tests - make tests self checking
  - Hand code examples and then compare results in the test
  - Great tests are easy to run and diagnose - you will be using them a lot - or should be
  - Most experienced developers worship tests and testers (or should!)
  - see any parallels to XP here?
- Break code into smaller pieces (extract method technique, later)
  - Look for variables in the fragment and their scope
    - Local variables, non modified, pass as parameter, modified return(so long as there is only one - recall java)
- Strive for small changes - since each change is small, errors easier to find

Refactoring Intro-2

- P15 - "any fool can write code that a computer can understand. Good programmers write code that humans can understand."
- Rename variables for clarity and then test again!
- A method should be on the object whose data it uses (strong cohesion, weak coupling) - move method technique (also rename it to be understandable in its new context). And, of course test.
- Find any reference to the old method and adjust reference to new method (why two steps? Because you are striving for small, careful steps)
- Remove old method … test
- In summary move/rename, fix references, remove old, testing at each step.
- Redundant variables? Replace temp with query. Test
  - Discourages long, complex methods

Refactoring Intro -3

- Use refactoring to optimize clarity, increase cohesion and loosen coupling. This sometimes requires adding code and additional loops. You cannot tell if it affects performance until the whole process is complete. Unlike frequent testing, performance tends to be holistic and it is best to wait.
  - It is also “apples to oranges,” if you add functionality and flexibility for future expansions of the application - then the performance hit is balanced by expandability and comprehensibility.

Chapter 2
Principles in Refactoring

- Refactoring (noun) a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- Refactor (verb) to restructure software by applying a series of refactorings without changing its observable behavior.
- Cleaning up code in an efficient, controlled manner
- Purpose is to produce code (and designs) that is/are easier to understand and modify - contrast with performance optimization which may make code harder to read (e.g., push to assembler)
- Does not change behavior - therefore should not change tests save for cases you missed

Principles-2

- Developing software involves refactoring ... Switch from adding function (and tests) to refactoring .
- Refactoring:
  - Improves design of software (or at least preserves design of software)
  - Makes software easier to understand
    - Not always thought of when you are trying to get code to work
    - Also a way of understanding the code - (careful here) if you test it and functionality is preserved you prove that you understand the code.
  - Helps you find bugs - clarity of code highlights them
  - "Helps you program faster, good design = rapid software development"

Principles - 3

- When to refactor:
  - When you are consistently wrestling with a design, for example consider the rule of 3 - the third time you do something similar, refactor
  - When you add a function
    - Refactor for understanding
    - Facilitates adding feature
  - When you need to fix a bug. This probably indicates a need to refactor since the software was not clear enough to see the bug initially
  - When you do a code review
    - A form of active reviews, you review by refactoring which aids in understanding
    - Keep such reviews small: one reviewer and the original author (XP)

Beck-Difficult code

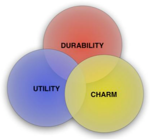
- Programs that are hard to read are hard to modify
- Programs that have duplicated logic are hard to modify
- Programs that require additional behavior that requires you to change running code are hard to modify - such changes may endanger existing behavior
- Programs with complex conditional logic are hard to modify - strive for simplicity

Principles - 4

- Convincing the luddite manager
  - Don't assume - other pressures
  - Active, more effective reviews
  - Quality
  - Schedule driven
  - Subversive

Issues with Refactoring

- Databases
  - Add an intermediate layer to isolate object (model) layer from database (model) layer
- Changing Interfaces
  - Refactoring does change the interface, e.g., rename method
  - Published vs. public interface - can't find and change all code that accesses it. For published you have to retain old and new interfaces until users can react to the change.
    - in java use deprecation facility to mark code as deprecated
    - Published interfaces are used too much? Makes refactoring difficult, change code ownership policy



Issues with Refactoring - 2

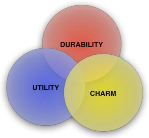
- Design changes difficult to refactor - central issues that may force you to redesign rather than refactor (actually refactoring at a higher level as we saw with Evans)
- When not to refactor
  - Rewrite from scratch instead (see design) - current code simply does not work
  - Dice a large software component into small components with strong encapsulation, then refactor or rebuild decisions are made a bit at a time for each of the now smaller components. (actually you already made refactor decision)
  - When you are very close to a deadline - then you go into “debt”

Refactoring and Design

- "With design I can think very fast, but my thinking is full of little holes" - Alistair Cockburn
- Programming is NOT a mechanical process
- Refactoring as the design process - XP
- Use CRC Cards - also may say a bit about size of systems
- Refactoring versus design, finding a reasonable solution rather than <u>the</u> solution (bit of a straw person)
- Simple vs. flexible (implies complicated design) - you only add flexibility where you need it

Refactoring and Performance

- Refactoring may make software run more slowly but will also make it more amenable to tuning.
  - Time budgeting - real time systems
  - Constant attention - everyone, everywhere concentrates on performance - narrow perspective - only small part of code usually affects performance -PROFILING
  - Refactor and use performance optimization stage late in the process, profile and tune in small steps
    - Smaller parts of refactoring add to finer granularity potentially adding to finer tuning
    - Slow first - much faster later



Chapter 3
Bad Smells in Code!

- Certain structures in code suggest refactoring - the rest of the chapter discusses these indicators (the book implies ordering):
- See smells/refactoring table on back inside cover of book
- Duplicated code is one category of bad smells, some types:
  - Same expressions in 2 methods of same class - Extract Methods (110- pg number)
  - Same expressions in 2 sibling subclasses -Extract Method (110) then Pull-Up Method(322) -- other subtleties if not exactly the same.
  - Duplicated code in unrelated classes - extract class(149) or perhaps code belongs in only one method in only one of the classes.

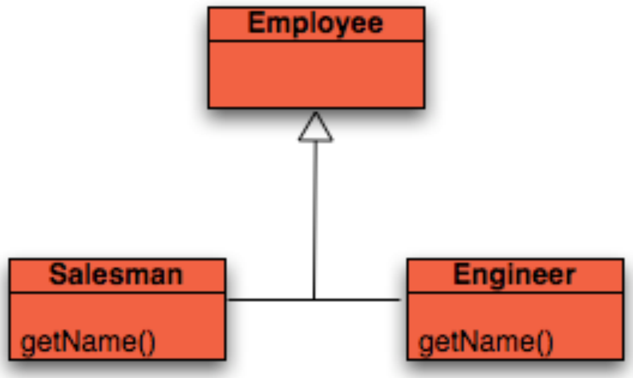
Extract Method (110)

```
void printOwing(double amount) {
  printBanner();
  System.out.println("name: "+ _name);
  System.out.println("amount: " + amount);
}
```

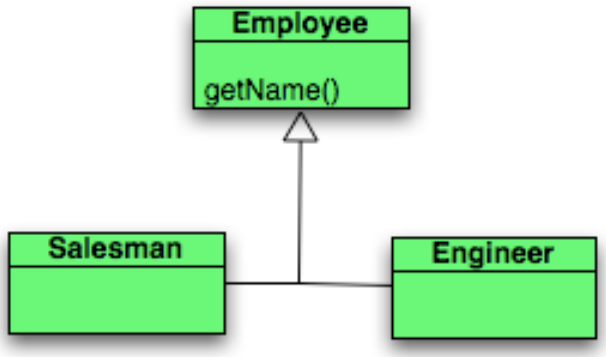
```
void printOwing(double amount){
  printBanner();
  printDetails(amount);
}

void printDetails (double amount){
  System.out.println("name: "+ _name);
  System.out.println("amount: " + amount);
}
```

Pull Up Method



Pull Up Method - 322



Long Method

- Long Method - OO metrics can detect this, the longer a procedure is the more difficult it is to understand, some types:
  - For typical long methods, use Extract Method (110), find parts of the larger method that go together and extract the method. (one key for clumps of code that can be extracted is descriptive naming)
  - Method with lots of parameters and temp variables -- attack this first
    - Replace temp with query (120)
    - Introduce parameter object(295)
    - Preserve whole object (288)
    - Still too many temps and parameters, replace method with method object (135)
  - Comments are a good indicator of clumps of code that can be replaced by a method
  - Conditionals and loops, sign for extraction - decompose conditional (238)

Large Class

- Large Class - one indicator is too many instance variables (suggests duplicated code)
  - Bundle a number of related variables and use extract class (149) or extract subclass (330)
  - Class with too much code is a breeding ground for duplicated code and chaos - eliminate redundancy in the class itself, tighten the code. Then extract class (149), extract subclass (330) and determine how clients hit code and extract interface (341) for each use.

Long Parameter list

- Long Parameter List - once taught as a good thing, rather than using global data. In OO parameter lists are (should be) smaller.
  - Replace parameter with method (292) when you can get data in one parameter by making a request of a known object
  - Replace a bunch of data with the data itself using preserve whole object (288)
  - If you have several bits of data with no logical object, use introduce parameter object (295)

Divergent Change

- When you must make changes in several places in a class to accomplish a change in the software functionality. For example, if you have to make a change in 3 methods every time you access a new database.
  - Use extract class(149) to put it all together.

Shotgun Surgery

- Related to divergent class, but you have to make a change that entails a lot of little changes to a bunch of classes -- hard to find, easy to miss one
  - Use move method (142) and move field (146) to place all changes in one class
    - If there is no class to put it in - create one
  - Inline class (154) can be used to bring a bunch of behavior together
- Divergent change is one class many changes, shotgun surgery one change, many classes

Feature Envy

- A method that is interested in a class other than the one it is in.
  - Invoking a bunch of get methods on another class to do its thing - use move method (142) and if only part of the method suffers from it use extract method (110) then move method (142)
  - If extracting data from several classes, where should it move? Where it is extracting most data!

Data Clumps

- Bunches of data that hang together in different places should have their own object
  - Use extract class (149) to accomplish this turning clumps of data into an object
  - Then use introduce parameter object (295) or preserve whole object (288) to slim them down by shrinking parameter lists and simplifying method calling
  - Then look for cases of Feature Envy that would suggest behavior to move into the new class

Primitive Obsession

- Blur distinction between built in and added classes in language, use this. Use small objects for small tasks, e.g., telephone numbers and zip codes.
- Refactorings for these are: replace data value with object (175), for data values and other refactorings that form objects in different situations -- small objects are good

Inappropriate Intimacy

- Basically violating information hiding
  - Move method (142) and move field(146) separate pieces
  - Use extract class (149) if classes do have a common interest
  - Use Hide delegate (157) to have another class act as go betweens
  - Sometimes subclassing can contribute so replace inheritance with delegation (352)

Popularity Poll

- High Runners: Move method (142), Extract Class (149), Inline class (154), Move field (146)
- Medium: Extract method (110), Introduce parameter object (295), Collapse hierarchy (344)
- Low: Rename Method (273), Preserve whole object(288), Replace inheritance with delegation (352), Hide delegate (157), replace type code with subclasses (223), replace type code with state/ strategy (227), introduce null object (260)

Building Tests

- As you might suspect, testing is important when refactoring
- Fixing a bug is easy, finding it is hard … substantial part of development is this activity
- Every method should have a test - make testing as painless as possible
- Test every time you compile
- Shades of XP, do the test before you do code, concentrates on interface, test should be so good/comprehensive that when the test works that indicates you are done coding

Testing-2

- Often when you refactor, you inherit a lot of code w/o self-testing
- The testing main on java - every class should have a main function that tests that class - can be tedious with lots of classes (but that too can be automated)
- Describes setting up a test environment using JUnit - open source
- Note that most of this is UNIT testing with some integration testing and system testing, especially if you are doing XP
- But if functional test/ users find something write a unit test(s) that exposes the bug.

Testing-3

- Testing should be risk driven, focusing on key aspects, versus write a test for every public method (some are trivial)
  - The key is to make writing tests a doable chore
  - P101 "Don't let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs"
- In testing look for ways to try to break code
- Even check if exception handling works (groan)
- Inheritance and polymorphism makes testing harder since there are many combinations to test
- Build a good bug detector and run it frequently -- share info with your testing team and if you follow a regular framework it should be easy for them to test
  - As part of your development team standards, you should include a common format for unit testing

Refactoring Catalog Categories

- Composing Methods: extract method, inline method, …
- Moving Features between objects: move method, move field, …
- Organizing data: encapsulate field, replace data with object, …
- Simplifying conditional expressions: decompose conditional, consolidate conditional expression, …
- Making method calls simpler: rename method, add parameter, …
- Dealing with Generalization: pull up field, pull up method, …

Composing Methods

- Composing methods - large bulk of refactoring, mostly methods that are too long
- Biggest issue with extract method (110) is local variables and temps

Extract Method

- Mechanics:
  - Create a new method and name it after the intention of the method
  - Copy extracted code from source method to new target method
  - Scan extracted method for vars that are local in scope to source method
  - See whether temp vars are used only w/in extracted code, if so make them temp vars of the new method
  - If these temp vars are modified by extracted code and if so see if you can treat it as a query and assign the result to the temp var concerned.
  - Pass into new method as parameters local scope vars that are read from the extracted code
  - Replace extracted code in source (original) with call to target method, You may be able to remove temp vars from original method, if now referenced solely by new method.
  - Compile and test

OO Metrics

- WMC is a measure of size of class, assumes larger classes are less desirable - usually a count of the number of methods
- RFC is number of methods in class + number of methods called by each of these class methods where each method is counted once
- LCOM is number of disjoint sets of methods of a class, any 2 methods in the same set share at least one local state variable, preferred value is 0, cohesion metric
- CBO is coupling metric, 2 classes are coupled if a method of one class uses a method or state variable of the other class, high values = tight bindings, undesirable
  - Gradations
  - Law of Demeter - methods of a class should only depend of top level structure of own class
- DIT is the distance of the class from the root of the tree. Language dependent
  - Strive for inheritance trees of medium height, not narrow and deep, not shallow and broad
- NOC is number of immediate descendants of a class, large number suggests improper abstraction

Last Words on these Refactorings

- Sometimes overall design is more appropriate
- The refactorings we just covered are on the whole locally oriented - The next few slides will cover refactorings larger in scope
  - One good aspect is that it makes metrics attractive
- You will become a better OO developer
- Judgement is still key

Big Refactorings - Chapter 12

- These are not as prescriptive
- They do not take hours - months - opportunistic redesign, do as much as you can when you can
- These are samples and not unsurprisingly the subject matter is dealing with inheritance and “OOness”

Tease Apart Inheritance

- Symptom: inheritance hierarchy doing 2 jobs at once -> turn into 2 hierarchies
- The trick is teasing apart these two jobs or dimensions the hierarchy currently represents - use graph paper to track the dimensions
- The problem is that this may be indicative of a larger design problem - gtv

Convert Procedural Design to Objects

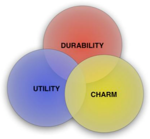
- Procedural -> turn data into objects, break up behavior
- Interesting process:
  - Take each record type and turn into dumb data object with accessors
  - Take all procedural code and put into single class
  - Take each long method and use Extract Method and others, then Move Method to appropriate dumb data class
  - Continue

Separate Domain from Presentation

- Shades of MVC - separate the view/presentation from the domain logic, the model.
- Contrasts with 2-tier style where data sits in data base and domain logic sits in presentation classes
- Process:
  - Create domain class for each window
  - If a grid, create a class to represent rows
  - Examine data on window classes, if used only for presentation leave, else if not displayed, move to domain object else if displayed, create duplicate observed data so it is in both places
  - Examine logic in presentation move domain logic to domain classes
  - (a key is work from the window, what the user sees rather than working from DB, what the developer sees)

Extract Hierarchy

- Symptom: class doing too much work as indicated by conditional statements (Swiss-Army-knife class) -> create hierarchy of class with each subclass representing a special case
- Tease it apart and this may call for a much more top-down design process - however, and this is part of the attraction of refactorings, you may never have enough time to do an overall redesign and refactoring gets you to an “approximation” gradually.



Refactoring Reuse Reality

- Refactoring should be seen as a middle ground
  - Makes design insights more explicit
  - Develops frameworks and extract reusable components
  - Clarifies software architecture
  - Prepares code to make additions easier
- Start with low level refactorings and only use a few of them
- Can be supported by automatic tools (because the refactorings are so low level)
- Brings the benefit of OO expertise in small steps - harder to get these things when you are not OO savvy.
- As refactoring becomes part of your routine it stops being considered (or feeling like) overhead to both you and your management.